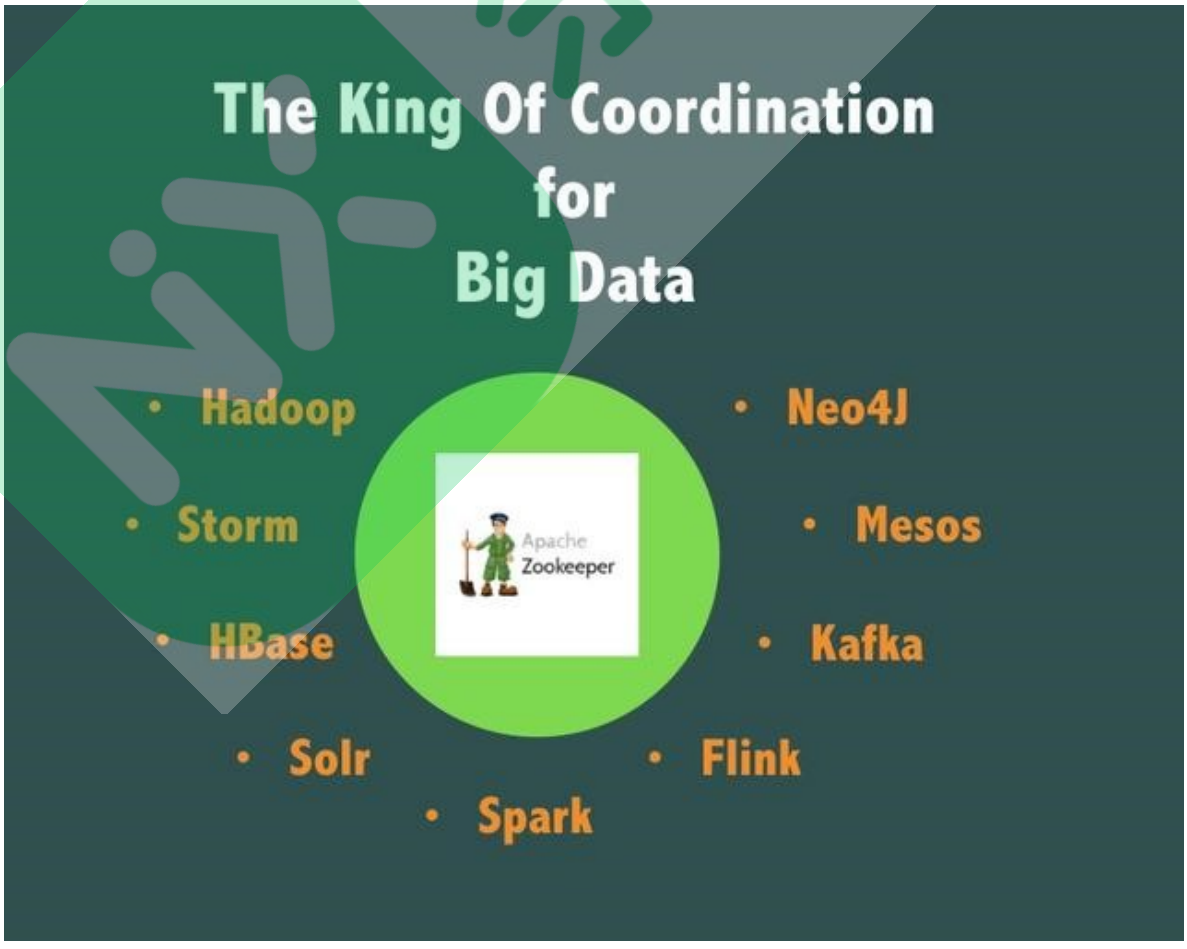
1. 课程介绍

ZooKeeper 是一款世界级的优秀开源产品，在大数据生态系统中 Hadoop、Storm、HBase、Spark、Flink、Kafka 随处都是 ZooKeeper的应用场景。特别是在粗粒度分布式锁、分布式选主、主备高可用切换等不需要高 TPS 的场景下有不可替代的作用。如果用一句话来评价ZooKeeper 的话，一定是“The King Of Coordination for Big Data”。



本课将以企业级真实案例驱动的方式，详细讲述架构师级 ZooKeeper 的架构设计、核心技术、以及在业界企业的真实应用实践。使得同学们能够真正掌握 ZooKeeper 这款世界级优秀的分布式协调产品。

2. 本次ZooKeeper内容大纲

Day01：ZooKeeper 基础设施和集群启动源码剖析

```
1、架构师级ZooKeeper总体架构设计深入剖析；
2、架构师级ZooKeeper基础设施源码深入剖析（序列化/持久化/网络通信/监听等核心机制）
3、架构师级ZooKeeper集群启动源码之集群启动流程深入剖析；
4、架构师级ZooKeeper集群启动源码之QuorumPeerMain启动深入剖析；
5、架构师级ZooKeeper集群启动源码之冷启动数据恢复等核心机制；
```

3. ZooKeeper 基础设施和集群启动源码剖析

在看任何技术源码的时候，都首先要搞清楚两件事：

```
1、版本选择
2、环境准备
```

3.1. ZooKeeper版本选择

你为什么需要看源码呢？两大看源码的需求支撑：

```
1、企业需求：你的项目遇到了困难，看源码解决
2、兴趣爱好 + 为了面试
```

zookeeper的大版本：

```
1、zookeeper-3.4.x  企业最常用，大数据技术组件最常用，基本维持在 3.4.5  3.4.6  3.4.7 这几个版本
2、zookeeper-3.5.x
3、zookeeper-3.6.x
```

最总结论：zookeeper-3.4.14.tar.gz，安装包就是源码包

ZooKeeper-3.5 以上，源码 和 安装包就分开了。

3.2. ZooKeeper源码环境准备

不需要过多的准备，准备一个 IDE，从官网下载源码包，然后直接用 IDE 打开即可！

```
1、准备一个IDE：IDEA
2、从官网下载源码包，IDEA去导入这个源码项目即可
3、稍微等待一下，maven去下载一些依赖jar
```

下载源码的方式：

```
1、从官网下载 zookeeper-3.4.14.tar.gz 安装包，该安装包直接包含源码
2、从 github 去拉取源码项目
```

3.3. ZooKeeper序列化机制

到底在那些地方需要使用序列化技术呢?

```
1、当在网络中需要进行消息，数据，等的传输，那么这些数据就需要进行序列化和反序列化
2、当数据需要被持久化到磁盘的时候。
```

ZooKeeper（分布式协调服务组件 + 存储系统）

任何一个分布式系统的底层，都必然会有网络通信，这就必然要提供一个分布式通信框架和序列化机制。所以我们在看 ZooKeeper 源码之前，先搞定 ZooKeeper 网络通信和序列化。

1、Java序列化机制

特点就是比较笨重：（除了实例的属性信息以外，还会序列化这个实例的类型信息）

```
class Student implements Serializable
```

使用 ObjectInputStream 和 ObjectOutputStream 来进行具体的序列化和反序列化。

2、Hadoop中的序列化：

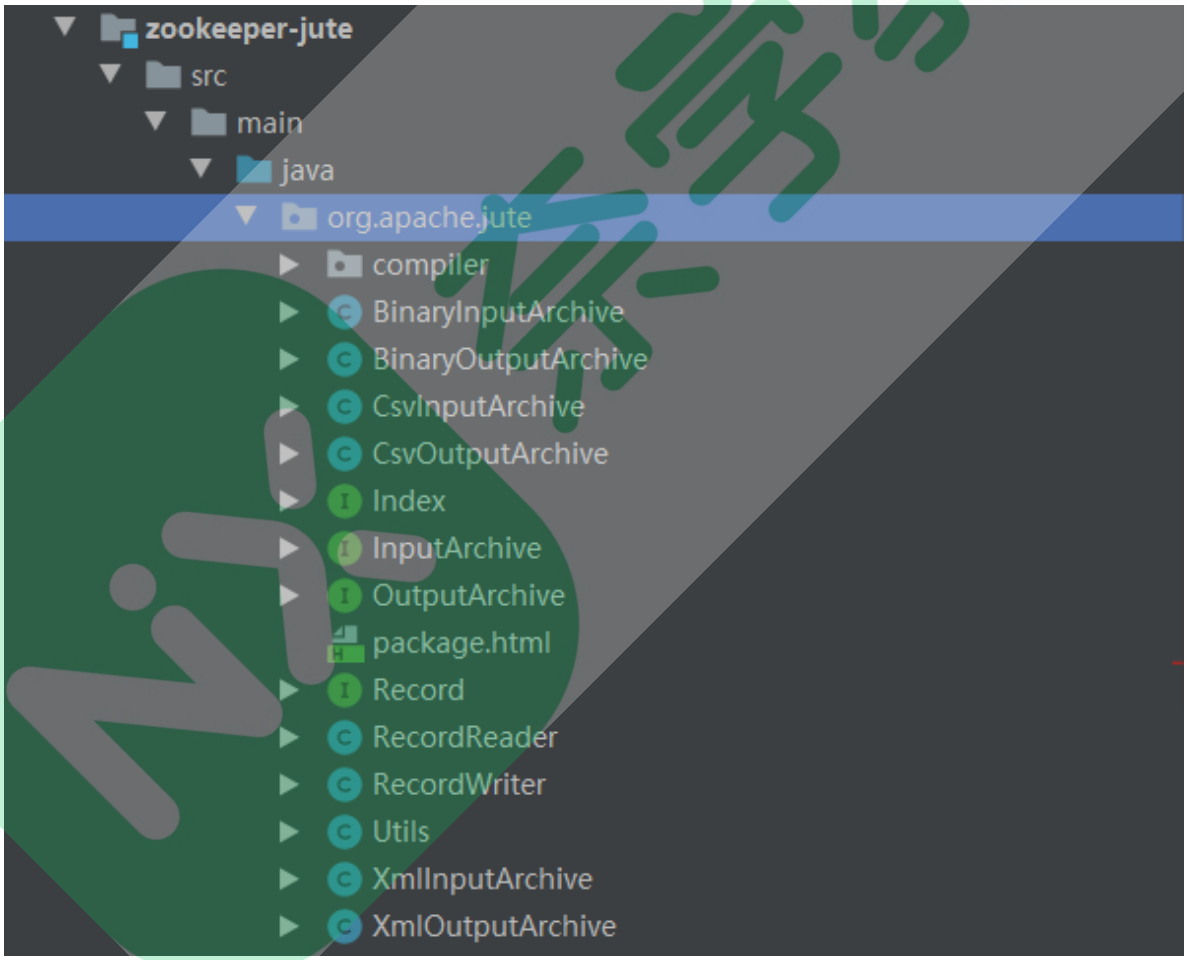
```
class Student implements Writable{

    // 反序列化
    void readFields(DataIn input);

    // 序列化
    void write(DataOut output);
}
```

3、ZooKeeper 中的序列化机制：

```
class Student implements Record{

    // 反序列化
    void deserialize(InputArchive archive, String tag){
        archive.readBytes();
        archive.readInt();
    }

    // 序列化
    void serialize(OutputArchive archive, String tag)
}
```

```
一个自定义类的实现，都是有多个普通数据类型的属性组成的!
class Student{
    private int id;
    private String name;
    .....
}
```

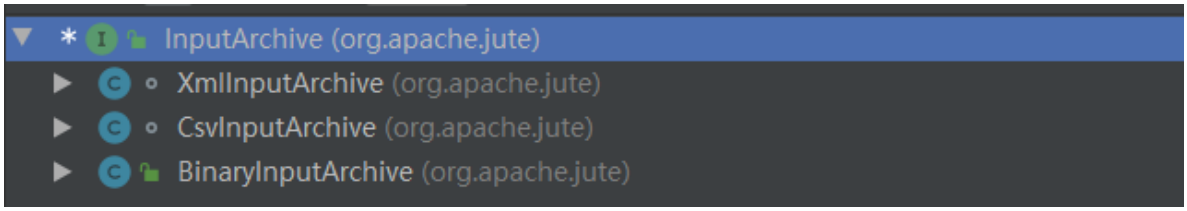
序列化的 API 主要在 zookeeper-jute 子项目中。



重点API：

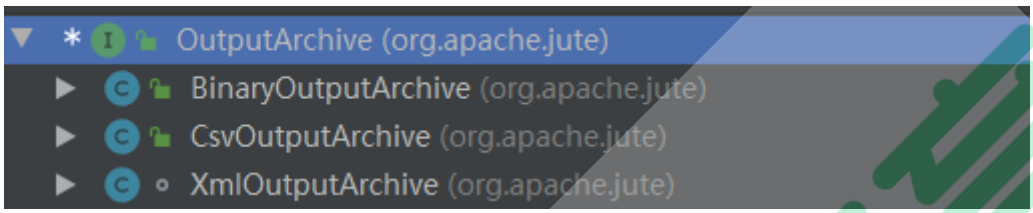
org.apache.jute.InputArchive：反序列化需要实现的接口，其中各种 read 开头的方法，都是反序列化方法

实现类有:



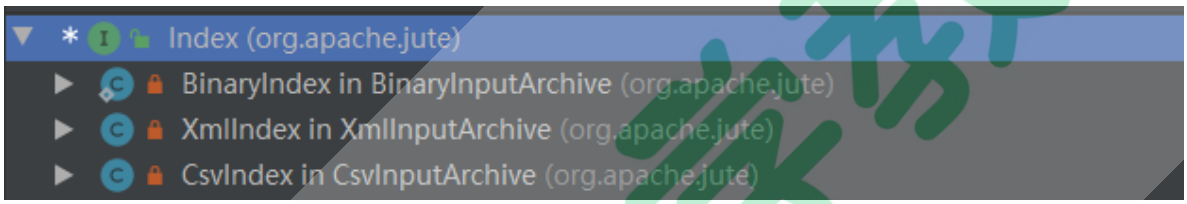
org.apache.jute.OutputArchive：所有进行序列化操作的都是实现这个接口，其中各种 write 开头的方法都是序列化方法。

实现类有：



org.apache.jute.Index：用于迭代数据进行反序列化的迭代器

实现类有：



org.apache.jute.Record：在 ZooKeeper 要进行网络通信的对象，都需要实现这个接口。里面有序列化和反序列化两个重要的方法

3.4. ZooKeeper持久化机制

ZooKeeper的数据模型主要涉及两类知识：数据模型 和 持久化机制

ZooKeeper 本身是一个对等架构（内部选举，从所有 learner 中选举一个 leader， 剩下的成为 follower）

```
1、每个节点上都保存了整个系统的所有数据(leader存储了数据，所有的follower节点都是leader的副本节点)
2、每个节点上的都把数据放在磁盘一份，放在内存一份
```

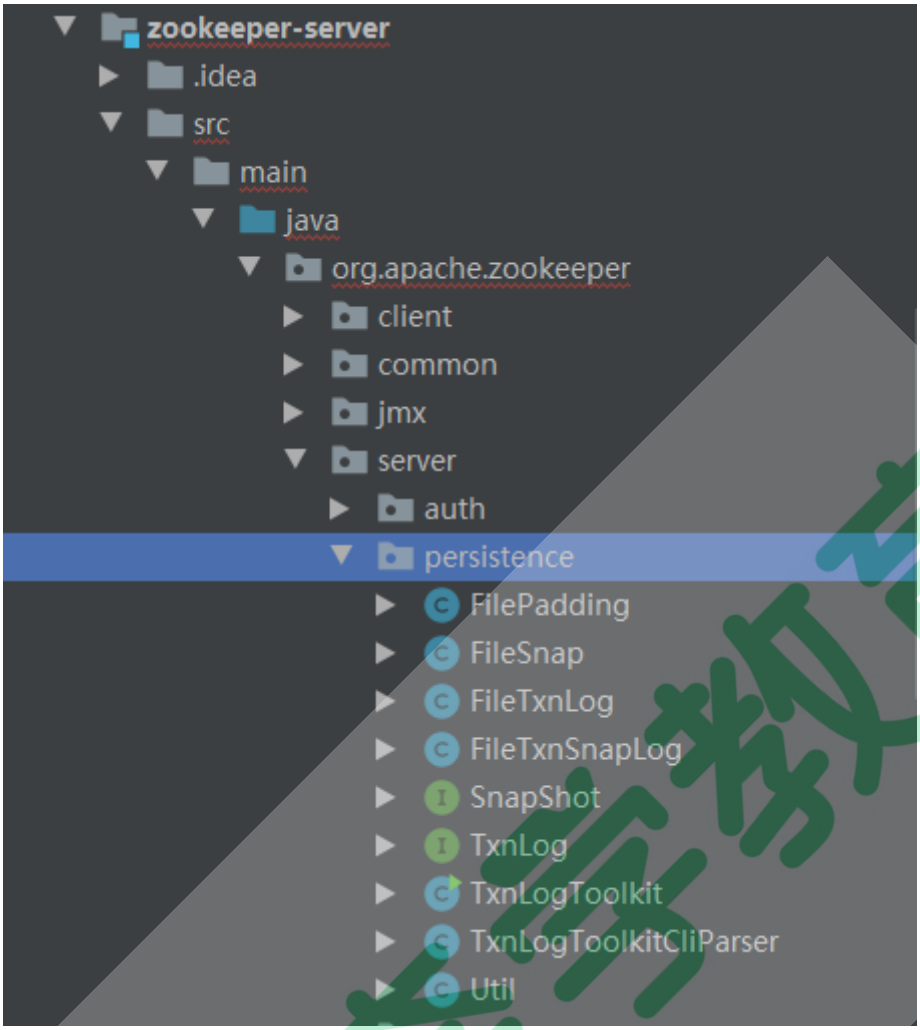
ZooKeeper的数据模型，抽象出了重要的三个API用来完成数据的管理：

```
1、DataNode       znode 系统中的一个节点的抽象
2、DataTree       znode系统的完整抽象
3、ZKDataBase     负责管理 DataTree ，执行 DataTree 的相关 快照和恢复的操作
```

关于 ZooKeeper 中的数据在内存中的组织，其实就是一棵树：

```
1、这棵树就叫做：DataTree  （抽象了一棵树）
2、这棵树上的节点：DataNode （抽象一个节点）
3、关于管理这个 DataTree 的组件就是 ZKDataBase （内存数据库：针对 DataTree 能做各种操作）
```

ZooKeeper 的持久化的一些操作接口，都在：org.apache.zookeeper.server.persistence 包中。



主要的类的介绍：

```
第一组：主要是用来操作日志的（如果客户端往zk中写入一条数据，则记录一条日志）
    TxnLog，接口，读取事务性日志的接口。
    FileTxnLog，实现TxnLog接口，添加了访问该事务性日志的API。

第二组：拍摄快照（当内存数据持久化到磁盘）
    Snapshot，接口类型，持久层快照接口。
    FileSnap，实现Snapshot接口，负责存储、序列化、反序列化、访问快照。

第三组：两个成员变量：TxnLog和SnapShot
    FileTxnSnapLog，封装了TxnLog和SnapShot。

第四组：工具类
    Util，工具类，提供持久化所需的API。
```

3.5. ZooKeeper网络通信机制

Java IO 有几个种类：(百度搜索；五种IO模型)

```
1、BIO JDK-1.1(编码简单，效率低) 阻塞模型
2、NIO JDK-1.4(效率有提升，编码复杂) 基于reactor实现的异步非阻塞网络通信模型
    通常的IO的选择：
    1、原生NIO
    2、基于NIO实现的网络通信框架：netty
3、AIO JDK-1.7(效率最高，编码复杂度一般) 真正的异步非阻塞通信模型
```

NIO 的三大API：

```
1、Buffer
2、Channel
3、Selector
```

ZooKeeper 中的通信有两种方式：

```
1、NIO，默认使用NIO
2、Netty
```

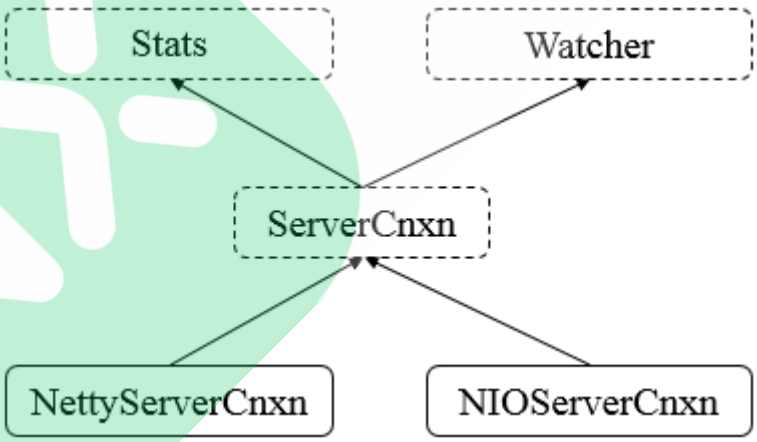
两个最重要的API：

```
ServerCnxn      服务端的通信组件
ClientCnxn      客户端的通信组件
```

关于客户端和服务端的一个定义：谁发请求，谁就是客户端，谁接收和处理请求，谁就是服务端

```
1、真正的client给zookeeper发请求
2、zookeeper中的leader给follower发命令
3、zookeeper中的followe给leader发请求
```

ServerCnxn：org.apache.zookeeper.server.ServerCnxn



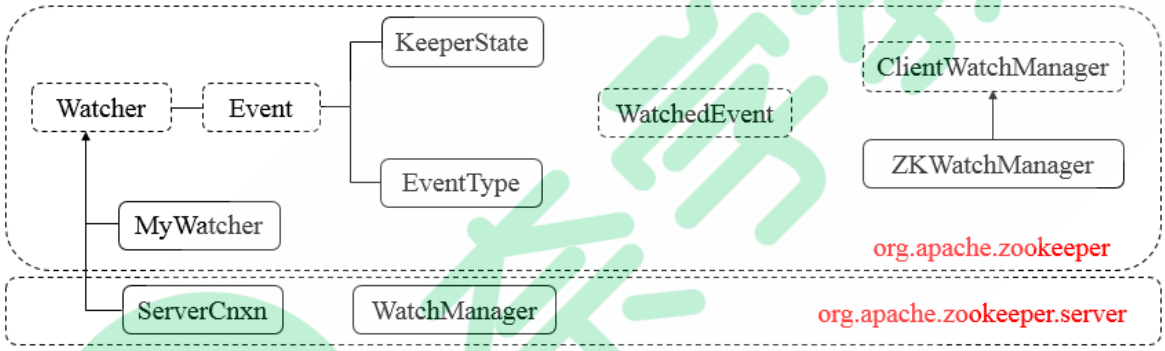
详细说明：

```
Stats，表示ServerCnxn上的统计数据。
Watcher，表示事件处理，监听器
ServerCnxn，表示服务器连接，表示一个从客户端到服务器的连接。
ClientCnxn，存在于客户端用来执行通信的组件
NettyServerCnxn，基于Netty的连接的具体实现。
NIOServerCnxn，基于NIO的连接的具体实现。
```

3.6. Zookeeper的Watcher工作机制

客户端的 Watcher 注册：

```
1、org.apache.zookeeper.ZooKeeper：客户端基础类、存储了ClientCnxn和ZkWatcherManager

2、ZKWatchManager：ZooKeeper的内部类，实现了ClientWatchManager接口，主要用来存储各种类型的Watcher，主要有三种：dataWatches、existWatches、childWatches以及一个默认的defaultWatcher

3、org.apache.zookeeper.ClientCnxn：与服务端的交互类，主要包含以下对象：LinkedListoutgoingQueue、SendThread 和 EventThread，其中outgoingQueue未待发送给服务端的Packet列表，SendThread线程负责和服务端进行请求交互，而EventThread线程则负责客户端Watcher事件的回调执行

4、WatchRegistration：Zookeeper的内容类，包装了Watcher和clientPath，并且负责Watcher的注册

5、Packet：ClientCnxn的内部类，与Zookeeper服务端通信的交互类
```



两条主线

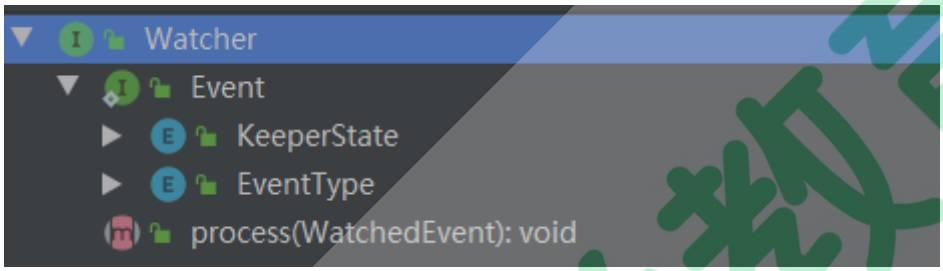
```
1、实现主线：Watcher + WatchedEvent
2、管理主线：WatchManager（负责响应：watcher.process(watchedEvent)） + ZKWatchManager（负责注册等相关管理）
```

```
interface Watcher{
    interface Event{
        enum KeeperState        链接状态
        enum EventType          事件类型
    }

    // 这就是回调方法（触发的事件：KeeperState, znodePath, EventType）
    void process(WatchedEvent event)
}

// 表示触发了一次监听事件的一个响应对象：链接状态 + znode节点路径 + 操作事件
class WatchedEvent{
    KeeperState state     会话连接的状态信息
    String path           znode节点的绝对路劲
    EventType type        事件的类型
}
```

组件说明：

Watcher，接口类型，其定义了process方法，需子类实现。
Event，接口类型，Watcher的内部类，无任何方法。
KeeperState，枚举类型，Event的内部类，表示Zookeeper所处的状态。
EventType，枚举类型，Event的内部类，表示Zookeeper中发生的事件类型。
WatchedEvent，表示对ZooKeeper上发生变化后的反馈，包含了KeeperState和EventType。
ClientWatchManager，接口类型，表示客户端的Watcher管理者，其定义了materialized方法，需子类实现。
ZKWatchManager，Zookeeper的内部类，继承ClientWatchManager。
MyWatcher，ZooKeeperMain的内部类，继承Watcher。
ServerCnxn，接口类型，继承Watcher，表示客户端与服务端的一个连接。
WatchManager，管理Watcher。

Watcher类组成：



WatchedEvent构成：

```java
/**
 *  A WatchedEvent represents a change on the ZooKeeper that a Watcher
 *  is able to respond to.  The WatchedEvent includes exactly what happened,
 *  the current state of the ZooKeeper, and the path of the znode that
 *  was involved in the event.
 */
@InterfaceAudience.Public
public class WatchedEvent {

    // 链接信息
    final private KeeperState keeperState;
    // 事件类型
    final private EventType eventType;
    // 事件发生的znode节点
    private String path;
```

Watcher 主要工作流程：

1. 用户调用 Zookeeper 的 getData 方法，并将自定义的 Watcher 以参数形式传入，该方法的作用主要是封装请求，然后调用 ClientCnxn 的 submitRequest 方法提交请求
2. ClientCnxn 在调用 submitRequest 提交请求时，会将 WatchRegistration(封装了我们传入的 Watcher 和clientPath)以参数的形式传入，submitRequest 方法主要作用是将信息封装成 Packet(ClientCnxn的内部类)，并将封装好的 Packet 加入到 ClientCnxn 的待发送列表中 (LinkedList outgoingQueue)
3. SendThread 线程不断地从 outgoingQueue 取出未发送的 Packet 发送给客户端并且将该 Packet 加入pendingQueue (等待服务器响应的Packet列表)中，并通过自身的 readResponse 方法接收服务端的响应
4. SendThread 接收到客户端的响应以后，会调用 ClientCnxn 的 finishPacket 方法进行 Watcher 方法的注册
5. 在 finishPacket 方法中，会取出 Packet 中的 WatchRegistration 对象，并调用其 register 方法，从ZKWatchManager 取出对应的 dataWatches、existWatches 或者 childWatches 其中的一个 Watcher 集合，然后将自己的 Watcher 添加到该 Watcher 集合中。

3.7. ZooKeeper的集群启动脚本分析

第一个问题；到底哪些源码流程我们需要关注呢?

```
1、集群的启动
2、崩溃恢复（leader选举）+ 原子广播（状态同步）
3、读写请求
```

第二个问题：到底从哪个地方入手看源码？入口

启动 ZooKeeper 的时候：

```
zkServer.sh start
```

底层会转到调用：QuorumPeerMain.main()

具体实现：见文档：ZooKeeper 启动脚本分析

3.8. ZooKeeper的QuorumPeerMain启动

大致流程：

```
# 入口方法
QuorumPeerMain.main();
    # 核心实现，分三步走
    QuorumPeerMain.initializeAndRun(args);
        # 第一步：解析配置
        config = new QuorumPeerConfig();
        config.parse(args[0]);
            Properties cfg = new Properties();
            cfg.load(in);
            parseProperties(cfg);
        # 第二步：启动一个线程（定时任务）来执行关于old snapshot的clean
        new DatadirCleanupManager(...).start()
            timer = new Timer("PurgeTask", true);
            TimerTask task = new PurgeTask(dataLogDir, snapDir,
snapRetainCount);
            timer.scheduleAtFixedRate(task, 0,
TimeUnit.HOURS.toMillis(purgeInterval));
                PurgeTxnLog.purge(new File(logsDir), new File(snapsDir), ...);
        # 第三步：启动（有两种模式：standalone，集群模式）重点关注集群启动，分两步走
        runFromConfig(config);
            # 服务端的通信组件 的初始化，但是并未启动
            factory = ServerCnxnFactory.createFactory();
            cnxnFactory.configure(....)
            # 抽象一个zookeeper节点，然后把解析出来的各种参数给配置上，然后启动
            quorumPeer = getQuorumPeer(); + quorumPeer.setXXX() +
quorumPeer.start();
                # 第一件事：把磁盘数据恢复到内存
                loadDataBase();
                    zkDb.loadDataBase();
                        # 冷启动的时候，从磁盘恢复数据到内存
```

```
                        snapLog.restore(....., ....)
                            # 从快照恢复
                            snapLog.deserialize(dt, sessions);
                            # 从操作日志恢复
                            fastForwardFromEdits(dt, sessions, listener);
                                # 恢复执行一条事务
                                rocessTransaction(hdr, dt, sessions,
itr.getTxn());
                # 第二件事：服务端的通信组件的真正启动
                cnxnFactory.start();
                # 第三件事：准备选举的一些必要操作(初始化一些队列和一些线程)
                startLeaderElection();
                # 第四件事：调用 start() 跳转到 run() 方法。因为 QuorumPeer被封装成
Thread 了
                super.start();
                    # 执行选举
                    QuorumPeer.run()
```

zoo.cfg 中的内容：

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/home/bigdata/data/zkdata
dataLogDir=/home/bigdata/data/zklog/
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
electionAlg=3
maxClientCnxns=60
peerType=observer
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in dataDir
#autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature
#autopurge.purgeInterval=1
server.2=bigdata02:2888:3888
server.3=bigdata03:2888:3888
server.4=bigdata04:2888:3888
server.5=bigdata05:2888:3888:observer
```

3.9. ZooKeeper的冷启动数据恢复

入口方法：QuorumPeer.loadDataBase();

大致流程：

```
# 入口方法
QuorumPeer.loadDataBase();
    zkDb.loadDataBase();
        # 冷启动的时候，从磁盘恢复数据到内存
        snapLog.restore(dataTree,...)
            # 从快照恢复
            snapLog.deserialize(dt, sessions);
                deserialize(dt, sessions, ia);
                    SerializeUtils.deserializeSnapshot(dt, ia, sessions);
                        dt.deserialize(ia, "tree");
            # 从操作日志恢复
            fastForwardFromEdits(dt, sessions, listener);
                 # 恢复执行一条事务
                processTransaction(hdr, dt, sessions, itr.getTxn());
                    # 恢复执行一条事务
                    dt.processTxn(hdr, txn);
                        # 创建一个znode
                        createNode 或者 deleteNode
```

详细内容见源码注释。

4. 总结

今天讲述的内容，主要是：ZooKeeper的基础设施组件和集群启动源码剖析。